ETL-0541

(2)

Parallel Vision Algorithm Design and Implementation 1988 End of Year Report

Takeo Kanade
Jon Adrian Webb

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

August 1989

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

ETL

Parallel Vision Algorithm Design and Implementation
1988 End of Year Report

Takeo Kanade and Jon Adrian Webb

CMU-RI-TR-89-23

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

August 1989

This research was supported by the Defense Advanced Research Projects Agency (DOD), monitored by the US Army Engineer Topograhic Laboratories under Contract DACA76-85-C-0002.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): CMU-RI-TR-89-23	5. MONITORING ORGANIZATION REPORT NUMBER(S): ETL-0541
6a. NAME OF PERFORMING ORGANIZATION: Carnegie Mellon University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: U.S. Army Engineer Topographic Laboratories
6c. ADDRESS (City, State, and ZIP Code): 5000 Forbes Avenue, Pittsburgh, PA 15213	7b. ADDRESS (City, State, and ZIP Code): Fort Belvoir, VA 22060-5546
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: Defense Advanced Research Projects Agency; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DACA76-85-C-0002
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Boulevard, Arlington, VA 22209-2308	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Parallel Vision Algorithm Design and Implementation 1988 End of Year Report

12. PERSONAL AUTHOR(S): Takeo Kanade & Jon Adrian Webb

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Annual	FROM 1/15/88 TO 1/14/89	August 1989	33

16. SUPPLEMENTARY NOTATION: Previous reports in series:
ETL-0467 May 1987 (for 1/86 to 1/87) End of Year Report for Parallel Vision Algor. Des.& Impl.
ETL-0513 Aug 1988 (for 1/87 to 1/88) Paral. Vis. Algor. Des. & Impl. 1987 End of Year Report

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Computer Vision, Systolic Processors, Benchmarks, Warp, Programming Languages, Parallel Computers, Image Processing,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Apply programming language has been extended to allow variable-sized image computations, and also to allow border mirroring, in which pixels accessed outside the borders are produced by copying pixels from the interior of the image. Implementation and design decisions are discussed. Apply and the Warp programming language W2 were used to implement the second DARPA image understanding benchmark. The results of this implementation are reported. Experience with this benchmark suggests a method for performing global image computations in a machine independent manner, using the divide and conquer model. Implications of this model for algorithms in the image understanding benchmark are discussed. It is shown that this model is capable of computing any algorithm in which data is accessed in a fixed order, regardless of the data values, and in which the final computation is reversible: that is, it produces the same results if the data values are reversed in order. Keywords: Artificial Intelligence

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: George Lukes	22b. TELEPHONE (Include Area Code): 202 355-2700; 22c. OFFICE SYMBOL: CEETL-RI-T

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Table of Contents

List of Figures

List of Tables

1. Introduction

This report reviews progress at Carnegie Mellon University (CMU) from January 15, 1988 to January 14, 1989 on research supported by the Defense Advanced Research Project Agency (DOD), monitored by the U.S. Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002, titled "Parallel Algorithms for Computer Vision Task B – Parallel Algorithm Design and Implementation." The report consists of an introduction, three detailed reports on specific areas of research, and a conclusion..

The three reports discuss the following areas of research:

- The Apply language, one of the major results of the parallel vision project at Carnegie Mellon, has been used by several DOD-sponsored sites in programming the Warp computer. Some of these sites requested several enhancements to the Apply language, including some enhancements that significantly extended its functionality for general use. These enhancements, specifically the ability to process varying-sized images with a single Apply program and the ability to mirror image borders, were done at Carnegie Mellon. We discuss here the enhancements and the design strategy behind them.
- Carnegie Mellon was a participant in the second DARPA Image Understanding Benchmark study, which compared the performance of a variety of architectures on a complete, integrated, image processing task. The performance and implementation of the benchmark on the Warp machine are presented. Both tightly-coupled data parallelism and loosely-coupled task-parallelism were exploited in order to get the best overall execution time on the task. We report the benchmark results and implementation methods and design choices here.
- Using the experience on this task and our previous experience from parallel vision, programming issues for low- and mid-level vision on parallel computers are examined. In particular, the problem of extending the Apply language (a machine-independent language for low-level computer vision) is considered. Extensions to Apply are proposed and it is shown how these extensions allow the efficient implementation of all of the low- and mid-level vision operations in the benchmark. The class of local and global operations that can be programmed with the extended Apply is considered. It is shown that all local and global operations that are *reversible* – which produce the same results when applied from the top down, or from the bottom up – can be programmed in the extended Apply.

2. Apply Enhancements

Two enhancements were made to Apply in 1988. The more significant of these was variable-sized image processing. The second was mirrored borders. The implementation of each of these will be discussed.

2.1 Variable-Sized Image Processing

Originally Apply and all Warp programs could process images and other data structures of a size specified at compile time. This design was chosen as a result of two observations: first, several hardware limitations of the Warp machine made it difficult to generate efficient code when the loop size in iterations was not known at compile time; second, in image processing images being processed usually come directly from a frame buffer, which produces images in only a few different sizes. Thus it is practical and efficient to restrict the image sizes.

However, as time went by, it became clear that there were good reasons to allow the image size to be specified at runtime. In processing large reconnaissance images in the ADRIES and SCORPIUS programs, the usual mode of processing was to restrict analysis to a limited area of the image, then take a region of the image from that area and process it further. The region of the image being processed could vary significantly in size. It was therefore requested that Apply be modified so that the image size could be specified at runtime. Similar considerations came from the NAVLAB project and other research projects at Carnegie Mellon. Clearly, making Apply capable of processing variable-sized images would be very useful.

In the summer of 1988, Han Wang, a graduate student visiting from the University of Leeds, modified Apply to allow it to process images whose size was specified at run time. This was an extensive modification that involved a number of design tradeoffs. We will present the background for this modification, the design tradeoffs encountered while doing it, and the implementation chosen.

2.1.1 Background

Several restrictions in the Warp architecture make it easier to implement fixed-size image processing, where the image size is known at compile time. These restrictions arose from a combination of trying to keep the design simple, and trying to take advantage of the restricted image sizes commonly used in image processing.

The Warp architecture restrictions that led to fixed-size image processing are:

- **Warp's deep pipeline.** Counting the time through the floating point adder or multiplier and the register file, an arithmetic operation takes seven cycles on Warp. These cycles must be software pipelined for best performance. In the first implementation of software pipelining, only loops with bounds known at compile time were implemented.
- **Warp's distributed control.** Three independent units must cooperate to control a Warp program: the external host processors, the interface unit, and the Warp cells themselves. Each of these must either be controlled by its own looping constructs or must get its control information from another of the units. Obviously, such control is easier to achieve if all loop bounds are known beforehand.
- **Data conversion restrictions.** A byte image is converted to floating point in the interface unit before being passed to Warp. Because of hardware restrictions this conversion must occur four bytes at a time. If the image width is not divisible by four, special precautions must be taken to pad it out and discard the extra data inside Warp.
- **Control restrictions in Warp's interface unit.** Originally, the interface unit did not include any provision for transferring data between the data and control paths. Thus, it was impossible to implement a run-time dependent loop on it.

These restrictions were removed as later versions of the Warp machine were developed. A control path from the external host to the interface unit was added so that the interface unit could loop under control of the external host, making it possible to implement a run-time dependent loop. The first two restrictions were software restrictions that were removed as the W2 compiler achieved a greater level of maturity. The data conversion restriction was handled

in the development of the variable image size Apply.

We took advantage of fixed size image processing to improve performance of Warp programs. A special input/output (I/O) model was implemented that would actually overlap communication with processing inside the Warp array.

This model evolved from an earlier design that did not overlap communication with processing. Early on, it was realized that processing data one row at a time, distributing that row across the Warp cells, had several advantages: (1) the image could be read in raster-order, making for faster access and easier programming; (2) it was easy to give each cell a contiguous group of columns, so that each processed a local region, for greater efficiency and programming ease; (3) local operators could be computed with little memory cost; and (4) it was easy for the cells to share columns at the border of their regions, which they would have to for local operators with extent.

This model was implemented with two macros, GETROW and PUTROW. GETROW read a row of data from the image and distributed it across the cells; PUTROW collected outputs from each cell and assembled them into one row, which was sent to the external host. These macros were easy enough to use that a W2 programmer, with care, could write image processing programs. They are still in use today.

These macros did not truly take advantage of Warp's systolic I/O facilities, however. Warp's independent functional units make it possible to, in one cycle, perform a floating point add and multiply, read both the X and Y queues, write both queues and perform a read and a write to memory. While accessing a message with GETROW or PUTROW, only queue and memory access occur. Systolic I/O is being abandoned in favor of message passing I/O, in the name of programmer ease.

While these macros worked well for compute intensive benchmarks, Warp's performance on I/O intensive benchmarks was inferior even to simple videorate processors. We therefore implemented a new macro that combined GETROW and PUTROW. It was called COMPUTEROW. It ran in three phases. In the first phase it would read in the data belonging to this cell, and store it in a memory buffer; it would also consume a portion of the output buffer representing the output this cell would produce. In the second phase, it would repeatedly compute one output and pass through several inputs and outputs to the following cells. In the third phase, it would send the output from this cell to later cells, and fill in the portion of the input buffer removed by its consumption of data in the first phase. The input and output buffers had to remain the same size so that each cell's code could be identical. The output buffer started out as all zero; these zeroes were generated by the interface unit on one pathway as it fed in the input data on the other.

A technical restriction in the W2 compiler made it necessary for the COMPUTEROW macro to generate multiple send and receive statements (instead of putting them inside a loop) in the second phase, while passing through inputs and outputs. The W2 compiler did not attempt to merge together into one microinstruction operations from different basic blocks. If the send and receive statements were in a loop, they would be in a separate basic block, and could not be merged together with the computation into one microinstruction. Since the send and receive statements were generated by the Apply compiler their number had to be known at compile time. Thus, the COMPUTEROW method could only be used when the image size was known then.

This macro took nearly full advantage of Warp's systolic I/O given the desire to process the image in raster order. In its central phase, it fully overlapped input, computation and output. The macro was somewhat difficult for the W2 programmer to use. But it led to respectable performance for Warp on many benchmarks.

In the implementation of Apply completed in the summer of 1987 by I-Chen Wu, the GETROW/PUTROW model was used first. Then code was written that would combine the code generated for GETROW and PUTROW into one code body that would effectively implement COMPUTEROW. This eliminated the ease of use problem noted with W2 programs; all Apply programs used the COMPUTEROW model, and achieved respectable performance without programmer difficulty.

2.1.2 Design Tradeoffs

A number of design tradeoffs were made when the variable-size image processing code was implemented. These included the I/O model within the array, the choice of where to do data type conversions, and the use of direct memory access (DMA).

The most important tradeoff was the I/O model. COMPUTEROW could not be used since the number of send and receive statements generated during the central portion of COMPUTEROW could not be known at compile time. Therefore, the old GETROW/PUTROW model was used. A modification had to be made for the PUTROW portion of the code, since the output buffer could not be initialized with zeroes at the interface unit. This was not possible because the number of zeroes that had to be generated at the interface unit was not known at compile time. However, since the Apply code used variable-bounds loops in the Warp code anyway, the output buffer could start at zero length before cell 0, and simply grow by the amount of data each cell added to the buffer after each cell. The code for GETROW did not change significantly. The result was that the code generated for variable-sized image processing was much simpler than the code for fixed-size image processing.

The resulting Warp programs accepted two numbers at run time; the image width and height. Inside the Warp array, these were used to calculate the number of columns for each cell, just as the calculation was done at compile time by the Apply compiler for fixed-size images. The cells then took the calculated number of columns from each row, passed them on to later cells, and computed the calculated number of outputs.

Conversions posed a second problem. For best efficiency, byte images must be converted to floating point for processing by Warp, which has two floating point units (add and multiply) and only one integer unit. This conversion could occur in the cluster processor, in the interface unit, or it could be done partly in the cluster processor and partly in the Warp cells. The tradeoffs between the three methods are as follows:

- **Conversion on the cluster processor.** This is the simplest method to program, and also the slowest. The input cluster processor needs merely to take each byte data and perform a floating point convert on it, then send it to Warp (and similarly for output). Since the cluster processor can access individual bytes there is no restriction on loop bound size, etc. However, converting a 512×512 image in this way can take as long as a second or more, creating a serious bottleneck since Warp can process such an image in a hundred milliseconds or less. Also, with this method byte images take longer to process that floating-point images, which do not need to be converted. This is counterintuitive; byte images should take less time since they are smaller.

- **Conversion in the interface unit.** This is the method used in processing fixed-size images; the interface unit was explicitly designed to do this. It is the fastest, but also the most difficult to program because controlling the interface unit is difficult. In particular, to deal with this control problem the W2 compiler makes it necessary that the number of data to be converted be known at compile time, by being defined inside a fixed-size loop.

- **Conversion partly in the cluster processor and partly in the Warp cells.** In this method each byte of image data is converted to a 32-bit word by being sign extended in the cluster processor, and then this word is sent through the interface unit without conversion. The 32-bit integer is converted to floating-point in the Warp cells. The conversion to a 32-bit integer is necessary because the Warp cells cannot access byte data. This method is only slightly more difficult to program than conversion in the cluster processor, and is faster since the floating point unit in the Warp cell is used instead of the one in the cluster processor, but it is not as fast as conversion in the interface unit since four times as much data is transferred between the cluster processor and the interface unit.

Given these considerations we first implemented variable-sized image processing using conversion within the cluster processors, in order to get a working implementation without worrying about performance. Later, when we implemented the DMA buffering code, we changed to use conversion in the interface unit.

DMA and conversion operations had similar restrictions. In order to use DMA, the number of data transferred in one DMA operation must be known at compile time; as with conversion in the interface unit, this restriction is enforced in the W2 compiler by requiring the send statements for the data to be transferred with DMA to be within a loop with bounds known at compile time. Because of this restriction we did not use DMA in the first

implementation of Apply with variable-sized images; DMA was added only later.

The DMA and conversion operation restrictions were finally addressed by having Apply split each row into a number of fixed-size messages, with the message size being known at compile time. (This message size is chosen to be a constant fraction—one-fourth of the maximum image width specified by the user, in the current implementation—in the Apply compiler). Apply rounds up the image width to be a multiple of this fixed size and then, for each row, it sends a number of fixed-size messages, with the send statements for each message being inside a fixed-size loop. Since the innermost loop is fixed in size, the W2 compiler uses DMA to transfer the data from the cluster processor memory to the interface unit and uses the interface unit to perform data conversions. The extra data transferred this way (to fill out the fixed-size message) are discarded by the first cell in the Warp array.

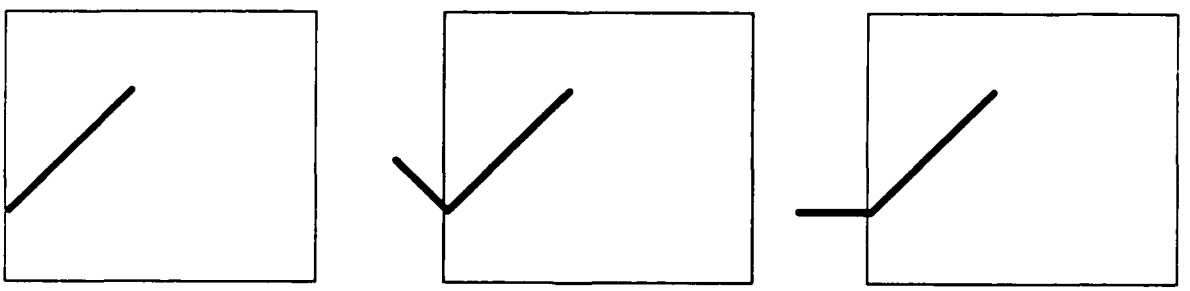
The resulting implementation of variable-sized image processing produces code that is efficient and somewhat simpler than the COMPUTEROW code for fixed-size image processing. The user specifies a maximum image size, and the Apply compiler generates code that can process any size image up to and including the maximum. Processing images of variable size is no harder than fixed-size image processing.

2.2 Mirrored Borders

When an Apply operator falls off the edge of the image, several different actions may be desirable. The simplest action is to fill the outside of the image with a constant value. From the beginning, Apply has supported this action. There are several other actions that are currently in use in image processing:

- **Border replication.** The last column and row are replicated outside the image.
- **Border mirroring.** Rows and columns from the interior of the image are replicated outside the image, so that image element (-1,0) is the same as element (0,0), element (-2,0) is the same as (1,0), and so on.
- **Wraparound.** The first column of the image is considered to be adjacent to the last column, and similarly for rows.

Each of these methods has advantages. Compared with filling with a constant, both border replication and border mirroring tend to give better results for operators that smooth or perform edge detection on the image: they will tend to give little false response near the edge of the image. Border mirroring will work well with smoothing even if the last row or column is anomalously large or small, while border replication would tend not to perform as well in this situation. However, border replication works better for some edge detectors, as was pointed out to the author by Gooitzen van der Wal, of SRI's David Sarnoff Research Laboratories. Consider the image shown in Figure 2-1(A). Border mirroring produces the image shown in Figure 2-1(B), while border replication produces the image shown in Figure 2-1(C). A corner detector, or an edge detector that also responded to corners, would incorrectly respond more strongly at the boundary of Figure 2-1(B) than at the boundary of Figure 2-1(C).



(A) (B) (C)

Figure 2-1:
(A) Image; (B) Image with mirrored borders; (C) Image with replicated borders

Wraparound is most useful when extremely low-frequency spatial components are being studied. The lowest frequency components of the image extend all the way across the image and can be detected with a low-frequency

operator with border wraparound.

Border mirroring was implemented in a straightforward way; we simply replicated the image top and bottom rows as they were sent to the Warp array (that is, the cluster processor sent these rows more than once), while the column replication was done inside Warp, in the first cell. We did not do column replication in the cluster processor since in general this would make it impossible to send the image to Warp a word at a time, since bytes would have to be repacked.

3. The Second DARPA Image Understanding Benchmark on Warp

The Second DARPA Image Understanding Benchmark addressed the issue of system performance on an integrated set of tasks that interact in a way typical of complex vision applications. The goal of the benchmark study was to gain a better understanding of vision architecture requirements that can be used to guide the development of future vision architectures.

In the First DARPA Image Understanding Benchmark [6] the problems to be solved were specified, but the algorithms were not given. This led to a great variety of different approaches to implementing each of the algorithms, which made it difficult to compare different architectures. The performance of an architecture depended on the algorithm chosen and the degree of optimization in its implementation as well as the performance of the computer. In this study both the problem to be solved and the algorithm were specified, and C code was given to solve the problem on a Unix computer.

In this benchmark, a complete image recognition system – the recognition of a two-dimensional mobile – was tested. This is in contrast to the first benchmark, where individual routines were tested. This approach has the advantage that overall timing is more realistic, since it measures the times that will be encountered when actually using the architecture to do recognition.

3.1 Benchmark Description

The benchmark involves the recognition of an approximately specified 2 1/2 D "mobile" sculpture composed of rectangles, given images from intensity and range sensors.

A complete description of the benchmark is available elsewhere [10]. We describe it here in outline form.

The problem is to recognize a two-dimensional mobile given a range and intensity image. The mobile is shown in Figure 3-1. It consists of a number of rectangular regions, connected by invisible threads, oriented randomly in planes normal to the line of sight. The rectangular regions have different solid grayvalues.

The input to the program is two 512×512 images. The first is a byte intensity image, the second is a floating-point depth image. The two images are registered, and are taken with parallel projection.

The program is given a set of 10 mobile models, and must choose the correct matching model from its set. There is only one correct match for each image. There are three factors that complicate recognition: the rectangles are allowed to rotate in the image plane around the invisible threads connecting them to the rest of the mobile, there are superfluous rectangles in the scene, and the depth image has added noise.

The processing in the task is required to proceed in the following steps:

1. Perform **connected component labelling** on the input intensity image. Since it is noise-free, it does not have to be filtered or thresholded first. Connected component labelling will correctly identify the rectangles in the image as long as they do not partially cover spurious rectangles of the same color, or are hidden by other model or spurious rectangles.
2. **Trace** and **compute k-curvature** of the borders in the labelled intensity image.
3. **Smooth** and perform **zero-crossing** detection in the **first derivative** of the border k-curvature. This detects corners in the intensity image.
4. **Count corners** of components. If there are three right angles in sequence, declare a rectangle.
5. **Median filter** the depth image. Either 3×3 and 5×5 operators could be used.
6. Perform a **Sobel** operator on the smoothed depth image to detect edges.
7. Using a depth-first graph matching technique, match the hypothesized rectangles against the model. Verify matches by **projecting** the images, using two techniques:

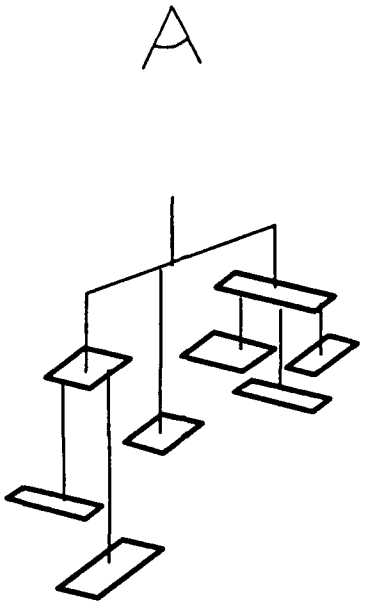


Figure 3-1: A Simple Mobile

a. **Depth** probe: within the hypothesized rectangle region, examine pixels; pixels deeper than the hypothesized rectangle count against it; pixels closer have no effect (since there could be a rectangle covering the hypothesized rectangle); and pixels at the right depth and of the right grayvalue count for the rectangle.

b. **Hough** probe: perform a hough transform on the region of the processed depth image within the rectangle and look for peaks at a predicted location.

8. **Paint** the detected model over the intensity image and output the result.

This recognition procedure is not necessarily the fastest method for solving the problem. In fact, as was pointed out by C. H. Chien of Carnegie Mellon, it is possible to "short-circuit" the benchmark. A simple histogramming operation applied to the depth image is sufficient to obtain the depth information of most of the rectangles, and these rectangles can be extracted using a multi-value thresholding technique in a later stage. There may be cases where a few rectangles have the same depth due to the resolution used in extracting depth information from the histogram of the depth image. These cases are rare and thus will not affect the result of matching.

The matching can be carried out in two steps: the initial matching and verification. The initial matching is trivial due to the assumption that all the rectangles are parallel to the image plane. This assumption allows us to impose an ordering on the rectangles in the mobile and to use "relative depth" as the *sole* feature for the initial matching. Note that the sizes, colors, positions and orientations of the rectangle have not been used up to this stage.

At the end of the initial matching, the depth of each rectangle of the mobile in the depth image can be easily determined, and this information can be used to guide the extraction of each rectangle in the depth image. As mentioned earlier, a simple thresholding technique is enough to determine which depth pixels are associated with each rectangle. The corners of that rectangle will be the ones with extreme values in x- or y- coordinates. Three corners can uniquely determine a rectangle (a small number of additional depth pixels may be required in the presence of noise). Edge detection, boundary following, or connected component labeling are unnecessary for extracting the rectangle. The extracted information is then used to verify the result of the initial matching.

Notice that the intensity image has never been used at all. It may become necessary to resolve some ambiguity in the depth image by using the intensity image, but this can easily be done by probing the intensity image at certain points, without needing to go through any sort of elaborate processing.

We now discuss each of the routines in turn.

3.2 Connected Component Labelling

The connected components algorithm was implemented by H. Printz [2, 5] of Carnegie Mellon. It uses a divide and conquer technique where slices of the image are first labelled independently on each cell, then the labels are unified by examining the borders between the cells. The merge is performed in a sequential pass; cell 0 merges its labels with cell 1, then cell 1 with cell 2, and so on. Finally, each cell passes its labelled image forward through the array where it is relabeled by the following cells, if necessary, to merge regions.

We did not attempt to use the recommended algorithm in this step, because we had existing code, and because the recommended algorithm would probably perform much worse than the Printz algorithm. The recommended algorithm propagates labels within the image without using an equivalence table. Each pixel is given a unique label based on its row and column coordinates, then labels are propagated from one pixel to another if both pixels have the same grayvalue in the original image. By performing repeated propagations in different directions, all pixels can be correctly labelled.

Depending on the order in which different directions are chosen, the worst case pattern can change, but for any choice of directions there is a pattern that requires this algorithm to make a number of steps on the order of the area of the image. (For example, if the different directions are always followed in the same order, a spiral pattern one pixel wide will a number of steps equal to at least half the image area). This is very poor performance in the worst case. However, it was not clear from the algorithm description whether worst case behavior was an issue. In the test images, all of the interesting regions were compact and rectangular. If we ignore the background, only a few passes of propagation were necessary.

In any case, this algorithm is better suited to large processor arrays with simple processors than to small processor arrays with powerful processors like Warp. The only operations that are performed in the recommended algorithm are comparison of pixel values, and assignment of the smaller of two integer values. This means that computers that provide these facilities, and no others (such as address calculation or floating-point arithmetic) in their hardware will be at an advantage for this algorithm.

The structure of the algorithm also makes it well suited to large processor arrays. Each step involves transfer of a label value and pixel value to a pixel's neighbor. The time for this step is $T \times I/P$, where T is the time for the computation for a single transfer and computation, I is the number of pixels in the image, and P is the number processors. The best case is obtained when $I=P$, so that there are as many processors as image pixels – a very large number. For Printz's algorithm, the execution time is $AP+B/P$, where A is the time to do a merge step, and B is the time to label the entire image serially. The best case is obtained when $P=\sqrt{B/A}$. With the parameters taken from the first DARPA IU benchmark, this gives P=16, which is much closer to the Warp machine's actual number of processors.

The connected components code consisted of 226 lines of W2 code. (Because the problem to be solved involves a global computation, only W2 could be used to program it on Warp). Of the 226 lines, 60 were in the labelling of the image locally, and 62 were in the computation of the global maps. The remaining 104 statements were declarations and I/O statements to read the image into and out of the Warp array.

A comparable program for a serial computer written in FORTRAN, CLAB from the Spider library [8], consists of 59 lines. The W2 code is thus nearly the same length (in its serial section, i.e., labelling the image slices individually) as the FORTRAN code. This reflects the similar level of the W2 and FORTRAN languages. The merge operations needed because of parallelism double the total code, and additional I/O statements and declarations double it again.

3.3 Tracing and Computing K-Curvature

This step could be broken down into two steps: (1) computing the direction numbers of boundary pixels in the labelled component image; (2) extracting the boundaries, smoothing them, taking their derivative, and detecting zero crossings. The first step could be done purely locally, since the direction number of a pixel depends only on the pixels in a 3×3 neighborhood around it. We implemented step (1) in Apply, and used the provided C code to perform the rest of the computation.

The Apply code was a straightforward translation from the provided C code. We show the Apply program In Figure 3-2 to illustrate its simplicity. For comparison we show the original serial C code in Figure 3-3. Note that the Apply program is shorter. Because the Apply compiler automatically handles border processing in a reasonable way, it is not necessary to include the `lok`, `tok`, etc. tests in the user program. Also, Apply makes image pixel references relative to the current position, which eliminates the `x` and `y` variables in the C program.

```
procedure set_chain(intensity : in array(-1..1,-1..1) of real,
                    ch        : out byte)
is
   cval: integer;
begin
   ch := 0;
   cval := intensity(0,0);

   if intensity(-1, 0) = cval then ch := ch | 1; end if;
   if intensity(-1, 1) = cval then ch := ch | 16; end if;
   if intensity( 0, 1) = cval then ch := ch | 2; end if;
   if intensity( 1, 1) = cval then ch := ch | 32; end if;
   if intensity( 1, 0) = cval then ch := ch | 4; end if;
   if intensity( 1,-1) = cval then ch := ch | 64; end if;
   if intensity( 0,-1) = cval then ch := ch | 8; end if;
   if intensity(-1,-1) = cval then ch := ch | 128; end if;

end set_chain;
```

Figure 3-2: Apply Code for Computing Direction Numbers

The second step of the code was executed using the provided Sun C code. This included a scan of the direction number image in order to trace boundaries, followed by three convolutions of the boundary vectors to detect corners.

This step could have been made faster by a Warp implementation. There were two reasons we did not use Warp: (1) The boundary tracing operation is difficult to do in parallel. The image could be split up among processors, or the boundaries could be distributed to processors. In the first case, a separate step is required to merge boundaries, which can be quite difficult. In the second case, locating the boundaries in the image and making sure that different processors have different boundaries requires a second component labelling pass over the image; and memory allocation is difficult, since each Warp cell does not have enough memory to store the whole image. (2) The boundary tracing could be done in parallel on the Sun with other steps in the system, leading to a lower overall time for the system if it was not done on Warp.

3.4 Median Filter

Depending on the problem, either 3×3 or 5×5 median filtering could be required. We had existing 3×3 median filter code already in WEB, which had been heavily optimized. The 5×5 code was written new for the benchmark.

3.4.1 3×3 Median Filter

The 3×3 median filter maintains three sorted column lists, only one of which must be updated as the window is shifted to the left. The three column lists are sorted according to their middle element, forming what we will call the "smallest," "middle," and "largest" list. The smallest two elements of the smallest list are smaller than at least

```
static unsigned char set_chain(intensity,x,y)
bmark_byte_image intensity;
int x, y;
{
register unsigned char ch;
register int cval,lok,tok,rok,bok;

  ch = 0;
  cval = intensity[y][x];

  /* set boundary flags */

  lok = x > COL_MIN;
  tok = y > ROW_MIN;
  rok = x < COL_MAX;
  bok = y < ROW_MAX;

  /* 4-connected boundary  & 8-connected boundary */

  if (tok) {
    if          (intensity[y - 1][x    ] == cval)  ch = ch |  1;
    if (rok &&  (intensity[y - 1][x + 1] == cval)) ch = ch | 16;
    }
  if (rok) {
    if          (intensity[y    ][x + 1] == cval)  ch = ch |  2;
    if (bok &&  (intensity[y + 1][x + 1] == cval)) ch = ch | 32;
    }
  if (bok) {
    if          (intensity[y + 1][x    ] == cval)  ch = ch |  4;
    if (lok &&  (intensity[y + 1][x - 1] == cval)) ch = ch | 64;
    }
  if (lok) {
    if          (intensity[y    ][x - 1] == cval)  ch = ch |   8;
    if (tok &&  (intensity[y - 1][x - 1] == cval)) ch = ch | 128;
    }

  return(ch);
}
```

Figure 3-3: C Code for Computing Direction Numbers

five elements of the 3×3 window, so they cannot be the median. Similarly for the largest two elements of the largest list. This leaves five elements; these are sorted. The median of these is the median of the 3×3 window.

In the Warp implementation of this algorithm the image is divided by columns into ten slices, each cell taking one slice, and the medians are calculated independently in each cell as the image is fed to the array row by row. This partitioning method wastes computation since each cell has to reinitialize the median filter for each row, as opposed to partitioning by row, where only one cell has to reinitialize the median filter for each row. It has the advantage of allowing computation of results on line, and requiring less memory per cell (which was the overriding factor when this code was implemented on the prototype Warp machine, which had only 4K words of memory per cell.)

3.4.2 5×5 Median Filter

While median filter can be formulated as a local operation, the provided C code used raster order processing to speed up execution time. This made it impossible to implement the code using Apply, which does not allow the programmer to take advantage of raster order processing. Therefore, the 5×5 median filter algorithm was a straight translation of the provided C code into W2. The image is divided into 10 horizontal slices, with each processor getting one slice. To compute the first median in a row the values in the 5×5 region are sorted and the 13^{th} largest is chosen. To compute the next median in the row five new values (the right side of the 5×5 region) are inserted into the sorted list, and the five values no longer in the region are removed from the list. The 13^{th} is then chosen as the

median for this window. This process repeats for the rest of the elements in the row.

The programming effort for this algorithm was minimal, since partitioning the image by rows is straightforward, and processing a row is done exactly the same way as in the C code. This is illustrated by the statement counts for the different parts of the code; 28 W2 statements were involved in programming the image I/O, and 52 statements were needed for the computation, versus 67 statements for computation in the C code.

3.5 Gradient Magnitude and Thresholding

The Sobel operator was a straightforward implementation of the provided C code. The Apply code for the Sobel operator is shown in Figure 3-4. The code development time for this program was only 10 minutes.

```
procedure grad(imagein  : in array (-1..1, -1..1) of real,
               threshsquared : const real,
               imageout : out byte)

is

   xderiv, yderiv : real;

begin

   xderiv := 0.25*imagein(-1,-1) +  0.5*imagein(-1, 0)
           + 0.25*imagein(-1, 1) - 0.25*imagein( 1,-1)
            - 0.5*imagein( 1, 0) - 0.25*imagein( 1, 1);

   yderiv := 0.25*imagein(-1,-1) +  0.5*imagein( 0,-1)
           + 0.25*imagein( 1,-1) - 0.25*imagein(-1, 1)
            - 0.5*imagein( 0, 1) - 0.25*imagein( 1, 1);

   if (xderiv * xderiv + yderiv * yderiv) < threshsquared
        then imageout := 0;
        else imageout := 255;
   end if;

end grad;
```

Figure 3-4: Apply Sobel Operator

3.6 Depth and Hough Probes

The depth and Hough probes were performed as part of a graph matching algorithm that ran on the Sun. The probe routines were compact loops that consumed the majority of the computation time for the graph matching. Moving these loops to Warp was straightforward, since they consisted of operations that accessed a rectangular area of the image, and performed a few simple tests on it.

3.6.1 Depth Probe

The depth probe took place in a bounding box aligned with the horizontal and vertical image axes in which a rectangle was supposed to lie. Pixels within the region were first tested to see if they fell within the supposed rectangle (which could have any orientation) and then, if they did, one of several counters was incremented depending on whether the pixel was deeper or closer than the supposed rectangle, or at the right distance and with the right intensity value.

Since the processing of each pixel was independent, there was no advantage in dividing the region in some large-grain fashion, such as splitting it into adjacent strips of columns. Moreover, there was considerable overhead in doing this, because the padding needed to make the image divide evenly into strips would vary for each region,

and would have to be computed at run time, since the bounding box size was determined then. Also, the loops distributing the image across the cells would have bounds that could be known only at run time, so that the compiler could not take advantage of known loop bounds to perform various optimizations, such as pipelining. Instead, the region was distributed across the ten cells by having each cell take every tenth pixel as the region was sent in raster order from the cluster processor. Once the entire region was sent, the cluster processor sent a series of sentinels that caused the cells to total and output their counters.

Code download was done only once per execution. It averaged 1.6 seconds (s) with a standard deviation of 1.3 s. The standard deviation of the code download time is high because the system software must verify, before each probe, that the code has already been downloaded and must also initialize the machine state. This time is included in the code download time, but varies with the number of probes. A linear regression model of code download time of the form $a+bN$, where N is the number of probes, gives a=185 milliseconds (ms), and b=11.3 ms. In other words, the initial code download takes 185 ms, and the incremental cost (in overhead counted as code download time) for each probe is 11.3 ms.

3.6.2 Hough Probe

While the image region to be processed could vary, the Hough space produced was fixed. Therefore, we chose to split the Hough space among cells, and have each cell process the entire image. This led to a straightforward W2 program, the inner loop of which is shown in Figure 3-5.

Each cell receives the pixel, and forwards it to the next cell. The cell then updates its portion of the Hough space, which is divided by theta values – each cell takes every tenth theta value. The calculation of `rho` is the same as in the C code. Once all pixels are processed, the cells concatenate the Hough space and output it to the host.

```
FOR i := strow TO endrow-1 DO BEGIN
  y := i - MID_ROW;
  FOR j := stcol TO endcol-1 DO BEGIN
    x := j - MID_COL;
    RECEIVE(l, x, ipixel); SEND(r, x, ipixel);
    IF ipixel <> 0 THEN BEGIN
      htp := 0;
      FOR theta_offset := 0 TO 80 BY 10 DO BEGIN
        theta := theta_offset + cellid;
        rho := fix((x*sin[theta] - y*cos[theta]) / RBINSIZE);
        IF rho < 0 THEN
          BEGIN posx := htp+9; posy := 0-rho; END
        ELSE
          BEGIN posx := htp; posy := rho; END;
        hough[posx][posy] := hough[posx][posy] + 1.0;
        htp := htp + 1;
      END;
    END;
  END;
END;
```

Figure 3-5: The inner loop of the W2 Hough program

The code was downloaded only once, with an average time of 900 ms, and a standard deviation of 748 ms. The standard deviation of Hough probe code download time is high for the same reason as with match strength probe. The linear regression model gives the initial code download time as 179 ms, and the extra time per probe is 13.9 ms. These times are consistent with those for match strength probe.

3.7 Paint Result

After the model was detected, it was painted over the input image and the resulting image was output. In the provided C code the paint rectangle routine was called repeatedly as the model graph was traversed. This would have considerably slowed the Warp implementation, since the overhead for program startup is significant. In our implementation, the rectangle descriptions are stored in a list, which is then passed to the Warp for painting all at once.

Two different implementations of the paint rectangle routine were tried. In the first Apply code was used. All rectangle descriptions were passed as parameters to the Apply routine. The Apply program checked to see if its pixel fell within the rectangle description, and if it did then it was painted. (The rectangle descriptions were ordered in the same order as the C code would have painted them. Hence the Apply code always produced the same result as the original C code, even when rectangle descriptions overlapped.)

In the second implementation, which was written in W2, each cell took one tenth of the rectangle descriptions. A row of the image was passed in, and each cell painted its portions of the rectangles onto that row, passing the painted row on to the next cell. As before, the rectangles were ordered so that painting happened in the same order as the original C code.

The second implementation was faster, mainly because some of the testing could be done on a per-row basis (namely, whether the row intersected with the circumscribing rectangle aligned with the image axes around the rectangle to be drawn). In the Apply code this had to be done on a per-pixel basis.

The time for this routine was 2.3 s with a standard deviation of 179 ms. Of this, approximately 97% (standard deviation 0.40%) of the time was spent in the Warp paint rectangle routine.

3.8 Exploiting Task-level Parallelism

Warp is programmed using the *attached processor* model, in which a general purpose host (a Sun workstation) runs the majority of the code, which is also the least time-consuming portion, and calls the Warp array to do time-consuming small portions of the program. In this benchmark, these portions of the code are those that access images. Almost every operation that touches all or a large portion of the pixels in the image runs on Warp. (The only exception is the boundary extraction code).

We take advantage of the intrinsic parallelism in this model at several points in the benchmark. Warp can run on its own, doing some image processing, while the Sun does other tasks in parallel.

Exploiting task-level parallelism in this way allows us to largely eliminate the overhead of reading in images during the benchmark, which would otherwise constitute a significant portion of the total benchmark time. Figure 3-6 shows where we were able to make use of task-level parallelism: that is, in "read depth image," "read model descriptions," and "extract strong cues." "Read depth image" involves reading the floating-point depth image from disk, and "read model descriptions" involves reading the model descriptions in, where they are stored in ASCII. Because the depth image is so large, it takes about one second to read; since the model descriptions are stored in ASCII, they also take about one second to read and translate into internal binary format.

Because the depth image and model data are read in parallel with a Warp computation, they do not contribute to the total benchmark time; their time is subtracted from the benchmark time. This effect is better than the normal speedup provided by faster hardware, which is only multiplicative.

While the depth image is being subjected to median filter and the Sobel operator on Warp, "extract strong cues" runs in parallel on the Sun. "Extract strong cues" is a complex procedure that is difficult to parallelize, and which would have required a lot of reprogramming to run on Warp. Moreover, since it could be run in parallel, its execution did not contribute to the total benchmark time (at least in the case when 5×5 median filter was used).

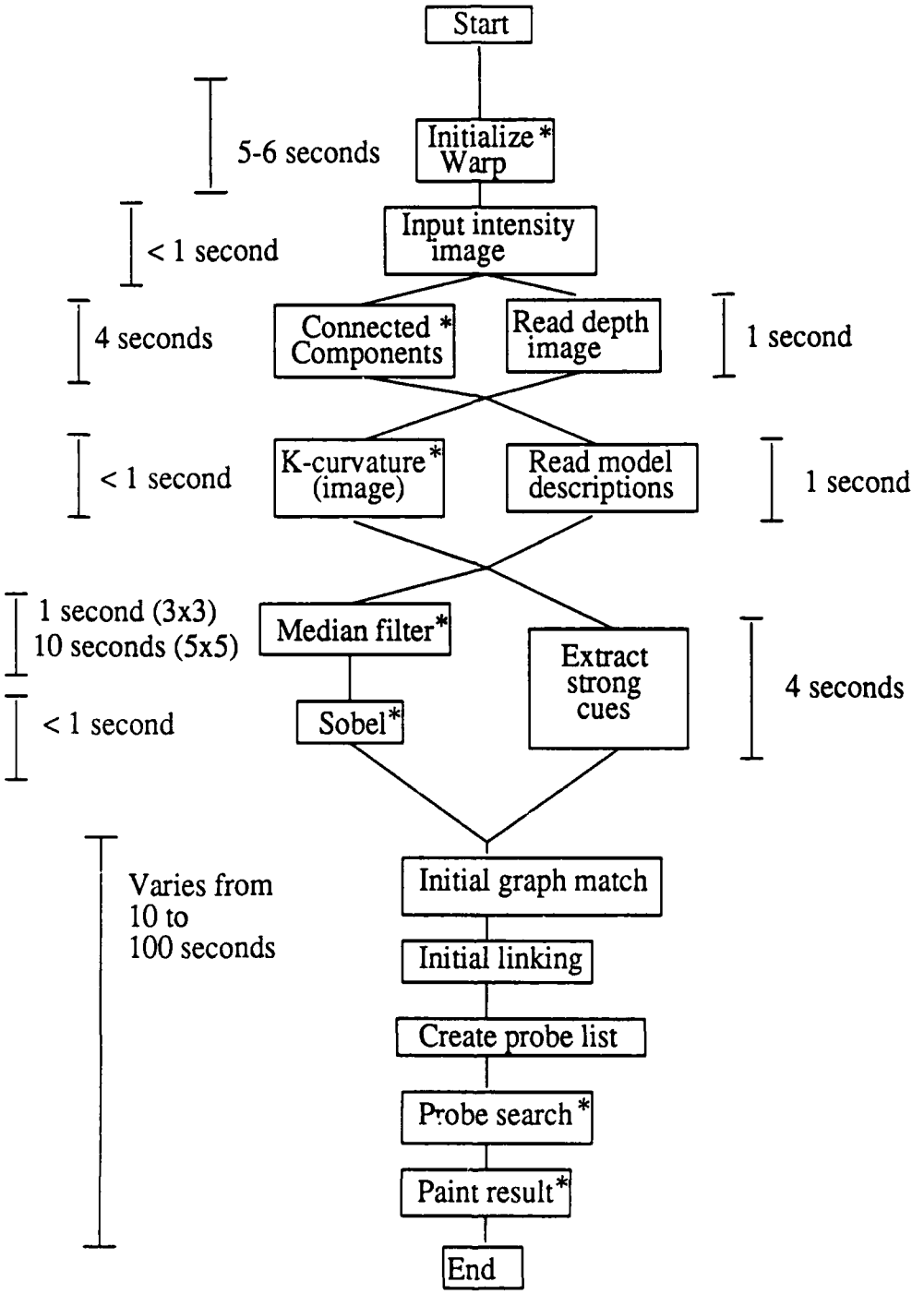


Figure 3-6: Task-level parallelism in the IU Benchmark

Exploiting task-level parallelism in this way is a powerful and general method of reducing total program time, because it does not rely on particular hardware features; instead, any routine that does not have dependencies with later routines can be executed in parallel. Task level parallelism is therefore much easier to exploit than data parallelism, which was exploited in the routines implemented on Warp; only minimal reprogramming is necessary. Unfortunately, there are only a few places in the benchmark where task level parallelism can be used. This is due to the general nature of recognition programs, where almost every reasoning step depends on those preceding it.

3.9 Performance Summary

Table 3-1 gives all the performance figures for Warp on the IU benchmark. The times given for indented lines are included in the first non-indented line above. Means and standard deviations of execution times are also given. All times are in seconds. All numbers are rounded to three significant digits.

The startup time for Warp programs was estimated to be about 25 ms, not counting code download time, which is given in the table.

Step	Sample	Test2	Test3	Test4	Test	Mean	Standard deviation
Task startup/initialization (initializing Warp)	5.76	5.96	5.88	6.00	6.04	5.93	0.111
Image input: intensity image	0.780	1.22	1.24	1.22	0.700	1.03	0.268
Image input: depth image	2.74	4.18	4.10	4.12	3.02	3.63	0.694
Model input	1.30	1.02	1.08	1.06	1.18	1.13	0.113
Label connected components	3.98	4.60	4.54	4.56	4.04	4.34	0.306
Warp code download time	0.400	0.420	0.440	0.420	0.420	0.420	0.0141
K-curvature	3.14	2.20	2.72	2.54	2.24	2.57	0.385
Warp code download time	0.260	0.260	0.260	0.240	0.260	0.256	0.00894
Smooth K curvature	1.38	0.780	0.980	0.900	0.640	0.936	0.279
First derivative of curvature	0.420	0.280	0.340	0.400	0.240	0.336	0.0767
Zero-crossing detection	0.320	0.120	0.140	0.220	0.060	0.172	0.110
Final corner detection (includes threshold and AND)	0.160	0.120	0.220	0.200	0.100	0.160	0.0501
Count corners for each component and threshold	0.0200	0.0400	0.0600	0.0600	0.0200	0.0400	0.0200
Convex hull of components	0.0200	0.0200	0.0800	0.0600	0.000	0.0360	0.0329
Test for sequence of three right angles	0.000	0.0200	0.0200	0.0200	0.000	0.0120	0.0110
Final rectangle hypothesis generation	0.0400	0.0200	0.0200	0.0400	0.000	0.0240	0.0167
3×3 Median filter		1.38	1.40	2.00		1.59	0.352
5×5 Median filter	10.7				8.70	9.70	1.41
Warp code download time	0.280	0.300	0.280	0.880	0.280	0.404	0.266
Sobel (includes conv., grad. mag., threshold)	0.480	0.720	0.940	0.920	0.480	0.708	0.225
Warp code download time	0.320	0.320	0.340	0.320	0.320	0.324	0.00894
Initial graph match	0.420	0.220	1.22	1.38	0.240	0.696	0.560
Rectangle matching	0.200	0.160	0.400	0.680	0.160	0.320	0.224
Initial linking	0.220	0.060	0.820	0.700	0.0800	0.376	0.358
Match extension	24.8	4.58	38.6	41.2	3.64	22.6	18.0
Match-strength probe	9.10	2.86	13.6	13.5	2.64	8.34	5.42
Total probes	91	20	247	239	23		
Window selection	0.0200	0.0200	0.240	0.180	0.0200	0.0960	0.106
Classification and counting	9.00	2.82	13.2	13.1	2.56	8.14	5.25
Time per probe	0.0989	0.141	0.0534	0.0548	0.113	0.0922	0.0380
Warp code download time	1.18	0.460	2.84	3.08	0.420	1.60	1.28
Hough probe	15.3	1.68	23.3	25.8	0.960	13.4	11.7
Total probes	58	5	97	95	3		
Window selection	0.0200	0.0200	0.120	0.0600	0.000	0.0440	0.0477
Hough transform	12.8	1.44	19.3	20.0	0.880	10.9	9.31
Warp code download time	0.840	0.260	1.06	2.08	0.260	0.900	0.748
Edge-peak detection	2.38	0.220	3.80	5.58	0.0800	2.41	2.36
Rectangle parameter update	0.0200	0.000	0.000	0.0800	0.000	0.0200	0.0346
Result presentation	2.60	2.52	2.24	2.26	2.26	2.38	0.171
Best match selection	0.0200	0.000	0.0200	0.0200	0.000	0.0120	0.0110
Graph traversal and image generation	2.54	2.46	2.16	2.18	2.20	2.31	0.178
Warp code download time	0.280	0.280	0.300	0.280	0.280	0.284	0.00894
Total execution time	52.9	29.9	65.8	69.3	29.4	[illegible]	19.1

Table 3-1: Performance of Warp on the IU Benchmark

4. Computing Global Image Operations

We now turn to the issue of programming low- and mid-level vision in a machine-independent manner. We have already developed (and used in the study in Section) a machine-independent language for local low-level image processing operations, called Apply. Use of this language in the benchmark led to fast implementations of some of the routines, compared to implementation of other routines with conventional tools. It would therefore be useful to be able to implement more of the routines with an Apply-like tool.

We propose specific extensions to Apply, then show how those extensions make it possible to program all of the low- and mid-level algorithms in this benchmark. We then consider what class of image processing operations can be programmed in the extended Apply. We will prove that it is the class of image processing operations that are *reversible*; that is, they produce the same result when applied to the image from the top down, or the bottom up.

In the implementation of the Warp routines discussed in Section , there is a strong difference between those implemented using W2 and those implemented using Apply. The Apply routines are shorter than the corresponding C programs, and generally took only a few minutes to program; the W2 routines are longer than the C routines, and generally required days of careful programming to get right. Here we examine the reasons for this difference.

The W2 programmer has to do several tasks that the Apply programmer avoids:

1. Input and output the data structures from Warp.
2. Sequence the operation across the input data structure. The C programmer must also do this step.
3. Combine separate portions of the output data structure into one structure.

The W2 programmer is also at one other important disadvantage compared to the Apply programmer; W2 code runs only on Warp, while Apply programs run on several machines, with more implementations underway. It is far easier to exchange Apply programs between machines, making Apply programs more useful as a medium for the exchange of ideas, and also Apply programs have a much longer useful lifetime than W2 programs. (For example, the Apply programs in WEB have been ported from Warp to iWarp and through two changes in Warp/W2 architecture without change, while the W2 programs needed or will need changes in all three cases.)

The W2 programmer, however, has two critical advantages compared with the Apply programmer: (1) W2 is a general language, and (2) it is possible to exploit raster-order processing in W2, giving greater efficiency. The first advantage is the reason that connected components and the Hough probe algorithm were written in W2; the second is the reason W2 was used for median filter and the paint result routine.

We now consider whether it is possible to extend Apply so that it can be made more general, and capable of exploiting raster-order processing, without losing the advantages listed above.

Let us make a list of requirements for the new, extended Apply:

1. It should be capable of raster-order processing for greater efficiency.
2. It should be capable of computing global image processing algorithms.

Considering the global image processing algorithms in this benchmark, we observe that they all order their processing in the following way:

1. The image is broken down into sections, one section per processor.
2. Each processor computes a global result on its section.
3. The global results are combined to create the global data structure for the whole image.

Our experience suggests that this *divide and conquer* approach to global image operations is useful and general. Therefore, we will use it in the extended Apply as the uniform paradigm for global image operations. We will add to Apply the ability for the programmer to define a special *combining* function; this takes the output of two Apply functions applied over areas of the image and combines them. For example, the combining function for histogram is

the operation of adding the two histograms. We also add a *termination* function, which is applied once to the final global data structure; this can discard intermediate results needed for the combining operations and produce the global output. In order to initialize the data structures that are being computed, we add an *initialization* function.

We will also add raster-order image processing to Apply. An important issue is whether this would make it more difficult to implement Apply (since restricting the programmer to order-independent operations was supposed to make Apply compilers easier to implement). Our experience suggest that it does not; all Apply compilers (except the one for SLAP [4]) process the image in raster order anyway. (Raster order processing makes sense only on processors that have much fewer processors than pixels. This is the reason it would be hard to implement efficiently on an architecture like SLAP. The same problem exists on other architectures, for example the Connection Machine [9].)

Raster order processing implies that there must be some way of initializing the computation, since raster order algorithms assume some previous state. The initialization function we added in order to compute global operations can serve this purpose. The current Apply function becomes the function that is applied repeatedly, in raster-order, across the image. Note that the presence of an initialization function allows us to start the raster-order processing anywhere in the image; this makes it possible to still process the image in parallel, with raster-order processing at each cell.

Raster order processing allows us to think in terms of processing regions of pixels. Because of this, we can add a useful restriction to the applications of the combining function; we require that Apply use it to combine global variables computed only over adjacent regions of the image. This makes it easier to implement many global image operations, for example connected components.

To summarize, extended Apply programs consist of up to four parts:

1. An initialization function, which can be run anywhere in the image.
2. A raster-order function, which is applied in raster-order across the image (wrapped around the borders of the image). The first execution of the raster-order function is guaranteed to be preceded by an execution of the initialization function.
3. A combining function, which combines the outputs of any two image regions to produce an output for the concatenation of the two regions. In order to make programming easier, we stipulate that the combining function will be applied only to adjacent "swaths" (groups of consecutive rows) of the image.
4. A termination function, which is applied once after the output of the entire image is computed.

It is not necessary to have all these parts in an Apply program; only the raster-order function is required.

To illustrate how the Apply compiler could use these functions on different parallel computers, let us call the initialization function I, the raster-order function R, the combining function C, and the termination function T. On a serial processor, these functions will be executed as follows on a $N \times N$ image:

$$I(), R(0,0), R(0,1), \ldots, R(0,N), R(1,0), \ldots, R(N,N), T()$$

Subscripts are used to represent the image pixel at the center of the window processed by the function.

On a multiprocessor where the image is broken down horizontally, each processor taking an adjacent set of rows, a similar program will be used, except that each processor will process only its rows, and the combining function will be applied to merge results from adjacent stripes. One processor's program might look like this:

$$I(i,0), R(i,0), R(i,1), \ldots, R(i,N), R(i+1,0), \ldots, R(j,N), C()$$

where the application of $C()$ merges this processor's results with the results from some other processor. The order of the combining operations depends on the interprocessor communication facilities.

If we partition the image vertically, each processor taking an adjacent set of columns (as in the Apply Warp implementation) we can still implement raster order processing by passing intermediate results between processors after each has completed processing its portion of the columns. The program on the first processor looks like this:

$I(i,0), R(i,0), \ldots, R(i,j), Send()$

where *Send*() sends the intermediate results from this processor to the next. Intermediate processors have this program:

$Receive(), R(i,k), \ldots, R(i,l), Send()$

and the last processor has this program:

$Receive(), R(i,k), \ldots, R(i,l), C()$

Processing the image in this way creates a pipeline of processors, staggered diagonally across the image. After processor 1 has started processing the first row, processor 0 goes on the process the second. This mapping would be efficient only on computers with low communications overhead, such as Warp.

There are several ways of merging regions computed on different processors. With only nearest-neighbor communications in a linear processor array, regions would be merged serially – i.e., the first region is merged with the second, this result is then merged with the third region, and so on, resulting in a tree of merge operations like that shown in Figure 4-1. On a machine that has more flexible interprocessor communications facilities, regions can be merged in parallel as shown in Figure 4-2.

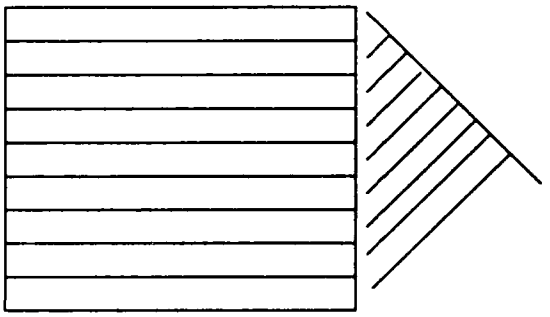


Figure 4-1: Merging results from adjacent image regions serially

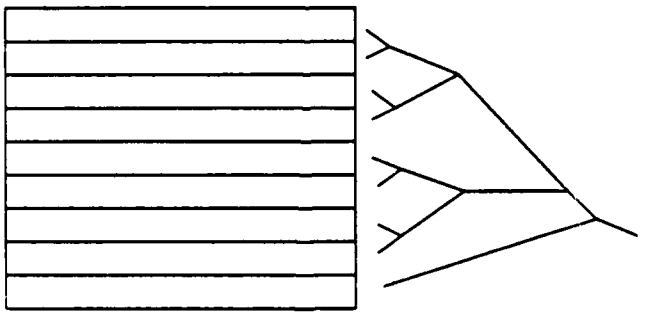


Figure 4-2: Merging results from adjacent image regions in parallel

The image can be broken down differently, for example by columns, or by allocating a square region to each processor. Dividing the image in such a way as to give each processor a complete row has the advantage that *Send*() and *Receive*() functions are unnecessary. However, breaking the image down in other ways also has advantages. Dividing the image by columns allows us to compute local Apply operations on-line; as each row is fed to the processor array, another row of results is computed. Dividing the image into squares allows us to use more processors efficiently, since Apply programs must duplicate processing at the region perimeter, and the rectangular region of a given area with the shortest perimeter is a square.

There is an important limitation in the divide and conquer when we try to reduce execution time by using more and more processors. Because of the overhead of the merge step, execution time decreases until a certain point, after which using more processors actually results in longer execution time. The number of processors that gives the

shortest execution time depends on the algorithm, the image size, and the interprocessor communication method. Let A be the time to merge two adjacent regions, and B be the time to process the entire image in raster order. With serial merging of regions as in Figure 4-1, the optimal number of processors is $\sqrt{B/A}$; on a two-dimensional array of cells with two merge steps (one horizontal and one vertical) the optimal number is $(B/A)^{2/3}$; and with the merge steps implemented as in Figure 4-2, each merge halving the number of regions to merge, the optimal number is $B\ln 2/A$. Table 4-1 gives the optimal number of processors for three algorithms: image sum, histogram, and connected components, all on 512×512 and 10,000×10,000 images. The time for 512×512 connected components comes from its Warp implementation in the First DARPA Image Understanding Benchmark; the other times are estimated.

We also give the "knee" of the execution time profile, where the benefit (increase in speedup) per unit cost (decrease in efficiency) is maximized [3]. Defining the efficiency of the parallel implementation on N processors as $E_N=T_1/(NT_N)$, where T_k is the execution time on k processors, this point is the k for which E_k/T_k is maximized. The point at which this happens is the most cost-effective number of processors to use. For a linear array, this number of processors at the knee is $\sqrt{B/3A}$; on a two-dimensional array, it is $(B/4A)^{2/3}$. There is no closed form solution for the number of processors with parallel merging; it is N in the equation $N=(B/A)\ln 2/(\ln N+2)$.

Algorithm	Image size	B/A	Serial merge		Two merge steps		Binary tree merge	
			Max	Best	Max	Best	Max	Best
Image sum	512×512	262,000	512	296	4100	1625	181,000	15,600
	10K×10K	100,000,000	10,000	5773	215,000	85,499	69,300,000	4,030,000
Histogram	512×512	1020	32	18	102	40	710	106
	10K×10K	391,000	625	361	5340	2120	271,000	22,500
Connected components	512×512	256	16	9	40	16	177	32
	10K×10K	5000	71	41	292	116	3470	430

Table 4-1: Global operations with different communications methods

Table 4-1 gives us some justification for allowing the programmer to assume that the image is divided only by rows when writing the combining function. For algorithms as complex as histogram, the overhead of the merge operation is great enough so that there is not enough parallelism available to require dividing the image in this way. The only cases where the available parallelism exceeds the number of rows are image sum with two merge steps or parallel merging, and histogram for binary tree merge (and even there the most cost-effective number of processors is near to or less than the image height). Even if we allowed image subdivision by columns, the Apply programmer and compiler could not make good use of it except with extremely simple merge operations. (In the absence of a combining function the Apply compiler can still divide the image by columns, without introducing any extra overhead, just as the current Apply does in its implementation on Warp.)

4.1 Benchmark Global Image Processing Operations in Apply

We now consider how the global image processing operations in this benchmark can be written in the extended Apply.

We will sketch the implementations of each algorithm and avoid detailed questions of their implementation in the new language. We order the algorithms in order of difficulty.

4.1.1 Median Filter

The median filter algorithm can be written using a raster-order function as follows:

- At the beginning of each row, the raster-order function sorts the elements in the window and stores them in a list. It then selects the median element from this list; this is the output. In preparation for moving the window to the right, the leftmost elements of the window are deleted from the list, and the

new rightmost elements are added.

Since the median filter algorithm does not compute a global result, there is no need for combination or termination functions.

4.1.2 Paint Rectangle

As supplied with the benchmark, the paint rectangle algorithm performs an important optimization; for a given rectangle, it determines from its bounding box whether or not the current row of the image intersects with the rectangle or not. Our extended Apply program can make the same optimization:

- At the beginning of each row the raster-order function determines, for the current row, whether or not it intersects each of the rectangles being painted. It then goes through each of the intersected rectangles in order (from most distant to nearest), determining if the current pixel lies within them, and repeatedly assigning the current pixel the color of the rectangle within which it falls. The result is that the current pixel gets the color of the nearest rectangle containing it.

4.1.3 Depth Probe

The depth probe operation as supplied with the benchmark processes only with the bounding box of the rectangle that is being probed. It computes a simple global operation on this rectangle:

- The initialization function zeroes the various counters: points too deep, points at the correct depth, and other points. It also zeroes a variable that keeps track of the sum of depths of points at the correct depth.
- The raster order function first determines if a pixel falls within the rectangle. If it does, it updates the appropriate counter, and adds its depth to the sum of depth variable if it is at the correct depth.
- The combination function simply adds together corresponding variables from the adjacent regions.
- The termination function calculates the average depth by dividing the sum of depth variable by the total number of correct points.

Note that the W2 program implemented for the benchmark used a different method of partitioning the image than any we have proposed for implementing extended Apply programs. The image was dealt out to the processors, one pixel at a time, with no concern for how the processors mapped onto the image; each of the ten processors simply took every tenth pixel. This was done because there is no dependence whatsoever between adjacent pixels in the code as supplied with the benchmark, and because it was easier to program.

However, the extended Apply makes it possible to propose a refinement in the algorithm, which can also be used in the paint rectangle routine. For each row, it is possible to calculate the starting and ending pixels in that row that fall within the rectangle; this computation can be done in the raster-order function. The resulting code avoids having to test every pixel for whether it lies in the rectangle. Since this test is difficult, involving several floating point operations, the resulting code is faster, and no more difficult to implement in the extended Apply.

4.1.4 Hough Probe

Hough probe performs a Hough transform on the Sobel-processed depth image to find rectangle edges. The region of the image processed is localized, but the Hough transform produced is not, although only a small portion of it is actually used. Using the divide and conquer model in this way has serious implications for memory usage. We present two different implementations of Hough probe in the extended Apply. In the first, we divide the image:

- The initialization function zeroes the Hough space, and also calculates the sine and cosine tables that are used to update it.
- The raster order function first determines if the pixel is non-zero. If it is, it increments all elements of the Hough space that are mapped by this pixel; this is a sine wave in the Hough array.

- The combination function sums the two Hough spaces from the adjacent regions to give the new Hough space.

There is no termination function because the final output of this algorithm is the Hough space. However, we could implement more processing in the Hough probe by including some of the probing for weak and strong edges in the termination function. This would make it unnecessary to output the Hough space.

This implementation of Hough probe requires each processor to store the complete Hough space; worse, when the two Hough spaces are combined one processor must store two of them. This is unfortunate, since in parallel processors memory is usually at a premium. The Hough space used here is 180×512; this would not fit, for example, in the Warp cell's 32KW memory, or in iWarp's 128KW memory.

The W2 code implemented for the benchmark divides the Hough space among processors. We can do this in the extended Apply, by treating the output Hough space as an image. In this way, updating the Hough space becomes similar to a graphics operation, with the output Hough space being the output image:

- The initialization function calculates sine and cosine tables as before. (Since the initialization function is run at the beginning of a row, and the Hough array is indexed as (ρ,θ), we must initialize the complete sine and cosine table for each row).
- The raster order function need only consider, for a given Hough pixel, what image pixels map onto it. This is a line in the image; the raster order function indexes along this line, and increments the Hough pixel for each non-zero image pixel.

There is no combination or termination function, because we have formulated Hough transform as a *local* operation by treating the input image as a *global* parameter. This algorithm could actually be implemented in the current Apply as it is formulated here. (It was not implemented in this way for the benchmark because of technical restrictions in the current Apply; global parameters must have dimensions that are known at compile time. The subimage processed by the Hough probe is determined at run time).

The second implementation of the Hough probe has an advantage when only a small part of the Hough space is actually needed, as here (where strong and weak edges are searched for in part of the Hough space). Only that part of the image that maps on to the corresponding portion of the Hough space will be examined, if we precede the raster order function by a test if the Hough pixel is needed. In the first implementation of Hough transform, there is no good way of avoiding this unnecessary computation.

In this implementation, the input image is stored at all processors, while the Hough space is distributed. If the input image is large, it may not fit. In this case, it is possible to process the input image in slices, incrementing the Hough space appropriately for each slice. This results in some unnecessary computation (because the line parameters for the image scan have to be recomputed for each slice), but it can make the memory space used for each pass arbitrarily small.

4.1.5 Connected Components

One of the assumptions underlying the divide and conquer model implemented by the extended Apply is the idea of data reduction. The output data structure of this level of vision is assumed to be smaller than the input, which is an image. This assumption, which is true of most vision algorithms at this level, is not true of connected components, since the output data structure is an image, just like the input. This assumption is also violated by Hough transform, but as we have seen it is not difficult to deal with this problem in Hough transform, because of the many sources of parallelism. It is harder to deal with it in connected components, however.

Connected components can be formulated in many different ways. These different methods have significantly different implications for the type of architecture that implements the algorithm most efficiently. The recommended algorithm for the benchmark is different from the algorithm we implemented on Warp; this is because the recommended algorithm is more suitable for a large processor array, while the algorithm we implemented is better suited for a small processor array like Warp. In this section we will examine both the algorithm we actually

implemented and the recommended algorithm from the point of view of the extended Apply.

First, let us consider the algorithm we actually implemented. It breaks down into three parts:

1. Split the image into separate parts, allocating one region to each processor, and label them separately, building an equivalence table for each part.
2. Examine the boundaries between the parts and merge the equivalence tables.
3. Apply the merged equivalence table to the image, producing the completely labelled image.

The final step is actually done by application of the first cell's equivalence table to the entire image, followed by application of the second cell's equivalence table, and so on. Doing the applications in this way allows us to make only a single forward merge pass (from the first cell to the last) across the equivalence tables, instead of having to make both a forward and backwards pass, and also avoids having to create a unified global equivalence table.

We will put the first two steps of the above algorithm into one extended Apply program, and put the second step into a second program, since each requires a pass over the image. The first two steps in connected components can be written as the following extended Apply program:

- The initialization function zeroes the equivalence table.
- If the new pixel is non-zero, the raster order function gives it a label, copying the label from the pixel to the left, left and above, above, or right and above, if any of them are non-zero, and otherwise assigning it a new label. If two of these pixels are non-zero and have different labels, it performs a union between the two labels and assigns the smaller of them to the new pixel. (A standard UNION-FIND algorithm [1] can be used, or see below.) The labelled pixels from the top and bottom boundaries of the image region are saved in a special buffer.
- The combining function merges the two equivalence tables by stepping along the touching boundaries, noting touching non-zero pixels, and performing a merge between the two equivalence tables.

There is an important optimization that can be performed in the raster-order function; because the image is two-dimensional, it is not necessary to perform a complete merge when two labels are merged. A region that is inside a second region can never be merged with any regions outside the second region. This optimization has been taken advantage of by several authors [5, 7].

The third step is a simple Apply program:

- The raster-order function simply looks up the pixel in the equivalence table and outputs its mapping.

This implementation of connected components suffers from one flaw, which was partially overcome in the W2 program implemented for the benchmark; the number of region labels can be potentially very large, as much as one-quarter (for 8-connected components) to one-half of the image pixels (for 4-connected components). This makes the equivalence table very large, in fact much too large to fit in one Warp cell's memory. In practice, however, (and in this benchmark) the number of components is not this great.

The W2 program addressed this issue by maintaining separate equivalence tables on each cell, which were never completely merged. As long as an individual cell did not need to label more components than would fit in its table, the algorithm would work. We cannot use this approach in the extended Apply, since it maintains an idea of a single value of all global variables for the entire image.

This restriction shows one of the assumptions behind the divide and conquer model that we are using in Apply; that is, that there is a reduction in the size of data being processed. Otherwise, the divide and conquer model becomes inefficient. In connected components this assumption is normally satisfied; but in extreme cases, it may not be, and connected components will have to be implemented directly in the target computer language, or in a different way.

Another way connected components can be implemented is as described in the algorithm supplied with the benchmark. In this algorithm, labels are simply propagated pixel to pixel, and no global equivalence table is used.

The simplest way of implementing this is as two ordinary Apply programs. The first initializes the image:

- Assign each non-zero pixel a label based on its position in the image.

while the second, which is executed repeatedly, propagates the labels:

- Copy the minimum of the adjacent non-zero pixels and the current pixel to the current pixel.

This implementation is extremely inefficient, however. Each pass moves a label only one pixel, and the longest connected component in an image can have a length on the order of the area of the image; so the running time of the algorithm is proportional to the image area. By taking advantage of raster-order processing, we can propagate labels all the way across one row or down one column in one pass, but this is not especially useful since propagating them in the other direction is just as hard as before.

4.2 Theoretical Restrictions of the Extended Apply

So far, we have taken a practical approach to understanding the class of image processing operations that can be computed in the extended Apply; we showed how several of the benchmark algorithms can be implemented in it. Now we consider the class of operations that can be implemented in extended Apply from a theoretical point of view.

Suppose we have a global operation $G()$ to be performed over an image $I=(a_0, \ldots, a_n)$. Let it be implemented by a series of applications of some raster-order operation $R()$:

$$G(I)=R(a_0,R(a_1,\ldots,R(a_n,\varnothing)))$$

$R()$ is the raster-order algorithm that is used to implement $G()$ on a serial computer. For example, if $G()$ is "calculate the histogram h()" then $R(a,h)$ is the operation "add 1 to h(a)," and the output of $R()$ is the new histogram.

On a parallel machine, we will execute a series of operations of $R()$ on different subsets of the image, then combine them somehow. We do this using a new function $C()$ such that

$$C(R(a_0,R(a_1,\ldots,R(a_{i-1},\varnothing))),R(a_i,R(a_{i+1},\ldots,R(a_n,\varnothing))))=R(a_0,R(a_1,\ldots,R(a_n,\varnothing)))$$

for any $0<i\leq n$. $R()$ is the raster-order function in extended Apply, and $C()$ is the combining function (we are ignoring the initialization and termination functions, which do not affect the proof.)

It turns out that for any local function $R()$, is is possible to define a function $C()$ that combines two outputs of $R()$ to produce the same result as repeated applications of a single instance of $R()$, so long as applying $R()$ to a sequence of data produces the same result as applying it in the reverse order.

The proof is as follows. Suppose we have the results of applying $R()$ repeatedly to two sequences A and B, and we want to compute the result of applying $R()$ to $A||B$. ($A||B$ is the concatenation of the sequences A and B). We write R^*A for the output of applying $R()$ to A, so we want to compute $R^*A||B$ given R^*A and R^*B. Given R^*B, we find (by enumeration, if necessary) a sequence C such that $R^*C=R^*B$. We then define

$$C(R^*A,R^*B)=R(c_0,R(c_1,\ldots,R(c_m,R^*A)))$$

Now we show that $C(R^*A,R^*B)=R^*A||B$. Consider the graph of applications of $R()$ that led to $R^*B=R^*C$, as shown in Figure 4-3. Below the point at which R^*B is computed, one branch of the graph leads down the application of $R()$ to the $\{c_i\}$, and another branch leads down the application of $R()$ to the $\{b_j\}$. We extend this graph upwards to the computation of $R^*A||B$ by applying $R()$ to the $\{a_k\}$. Now, since $R()$ is is reversible, we can invert the graph and get the same result $R^*A||B$ below c_0. But this result is exactly what we have defined as $C(R^*A,R^*B)$. This completes the proof.

This proof does not show that $C()$ can be computed efficiently. In extended Apply, the programmer is responsible for defining $C()$ reasonably. Note that if the programmer's definition of $C()$ computes the same result regardless of the order of application of $R()$, then $R()$ does not have to be reversible. Also, $C()$ may not compute the same result,

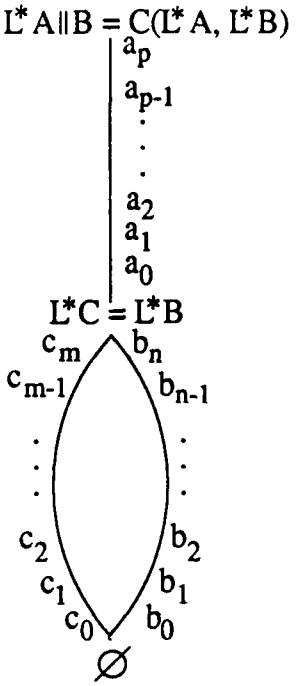


Figure 4-3: Graph of applications of *R*() to compute $R^*A||B$

but different results may be equivalent for the programmer's purpose. For example, floating point computations are not order independent, but generally the errors are small enough so that this does not matter. Or different orders of evaluation may calculate different results, but these might all be the same for the programmer's purpose – for example, the construction of the equivalence table in the connected components algorithm.

As we have seen, most global image computations can be formulated in this manner. Those that cannot are image processing operations that depend on processing the image in a particular order, such as some half-toning algorithms, the forward and backward pass in the two-pass grassfire algorithm, and certain region merging operations.

5. Conclusions

In this report we ahve seen how the Apply programming language has become a focus of the parallel vision effort. Developing Apply has yielded great benefits. It has directly benefitted the development of new parallel vision algorithms; it has extended the longevity and portability of our code; it has yielded new insights in parallel computing related to the implementation of important computer vision algorithms in parallel; and it has led us to a deeper understanding of the clases of computer vision algorithms and the scientific issues encountered in implementing them in parallel.

Section illustrates the direct benefits to the implementation of parallel vision algorithms. It is now possible to compile Apply programs to process images with the image size specified at run time, and to do sophisticated mirrored image border processing. The difficulties encountered in implementing such code by hand on Warp are considerable; but the Apply programmer encounters no such difficulties – in fact, the original Apply code developed in the second year of the parallel vision project can now be recompiled for variable size image processing without change. This is a significant enhancement of the effective capabilities of the Warp machine.

We have also seen how Apply aided longevity and portability of out parallel vision code. Apply code has now been mapped onto a wide range of parallel computers; the identical code can be compiled for existing and future machines. The code developed in the second year of the parallel vision project will be used on a wide range of projects for years to come.

Apply has also yielded new insights in parallel computing. By eliminating the local low-level vision algorithms from consideration, it led us to focus on global operations, which are very diverse. This focus is evident in our implementation of the Second DARPA Image Understanding Benchmark in Section .

In our implementation of the benchmark, we explored a number of programming issues: choosing which routines to run on Warp, and which to run in parallel (when possible) on the Sun; choosing which to write in Apply, and which had to be written in W2; and choosing how to partition the data in the W2 programs.

We now believe it is possible to confront the issue of programming mid-level vision routines in largely machine-independent manner. We propose to extend Apply using the divide-and-conquer programming model to do this.

In Section we showed how the DARPA IU Benchmark routines can be implemented in this model, and how flexible the model is for experimenting with different possible mappings of the routines. The theoretical basis of the model was explored, and it was shown that a general class of global image processing operations – those that are reversible – could be implemented with it.

The final result of our work is encouraging; we developed a new tool for parallel programming, and showed how this tool could directly aid programming of existing computers and portability and longevity of code. We have identified a new class of important computer vision algorithms that are amenable to implementation with an extended version of this tool. We now embark on the further development of this tool, together with its practical use in existing and future computer systems.

References

[1] Aho, A., Hopcroft, J.E. and Ullman, J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1975.

[2] Deutch, J., Maulik, P. C., Mosur, R., Printz, H., Ribas, H., Senko, J., Tseng, P. S., Webb, J. A., and Wu, I-C.
Performance of Warp on the DARPA Architecture Benchmarks.
In *International Conference on Parallel Processing for Computer Vision and Display.* Leeds, England, January, 1988.

[3] Eager, D. L., Zahorjan, J., and Lazowska, E. D.
Speedup versus efficiency in parallel systems.
IEEE Transactions on Computers 38(3):408-423, 1989.

[4] Fisher, A. J. and Highnam, P. T.
Communications, scheduling, and optimization in SIMD image processing.
In *Computer Architecures for Pattern Analysis and Machine Intelligence.* IEEE, Seattle, Washington, December, 1987.

[5] Kung, H.T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

[6] Rosenfeld, A.
A Report on the DARPA Image Understanding Architectures Workshop.
In *Image Understanding Workshop*, pages 298-301. DARPA, Los Angeles, California, February, 1987.

[7] Schwartz, J., Sharir, M., and Siegel, A.
An efficient algorithm for finding connected components in a binary image.
Technical Report 154, New York University Department of Computer Science, February, 1985.

[8] Electrotechnical Laboratory.
SPIDER (Subroutine Package for Image Data Enhancement and Recognition).
Joint System Development Corp., Tokyo, Japan, 1983.

[9] Tucker, L. W., and Robertson, G. G.
Architecture and Applications of The Connection Machine.
Computer Magazine 21(8):26-38, August, 1988.

[10] Weems, C., Riseman, E., Hanson, A., and Rosenfeld, A.
A Computer Vision Benchmark for Parallel Processing Systems.
In Karashev, L. P and S. I. (editor), *Third International Conference on Supercomputing*, pages 79-94. International Supercomputing Institute, Inc., Boston, MA, May, 1988.